UNIVERSIDAD NACIONAL
SAN AGUSTIN

FACULTAD DE PRODUCCION Y SERVICIOS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Informe sobre KD Tree

Estructuras de Datos Avanzadas

EDUARDO ANTONIO SANCHEZ HINCHO
LUIS GUILLERMO VILLANUEVA FLORES

AREQUIPA
2019

1. Introducción

En ciencias de la computación, un Árbol kd (abreviatura de árbol k-dimensional) es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de k dimensiones. Los árboles kd son un caso especial de los árboles BSP. Un árbol kd emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol kd, desde el nodo raíz hasta los nodos hoja, almacenan un punto. .

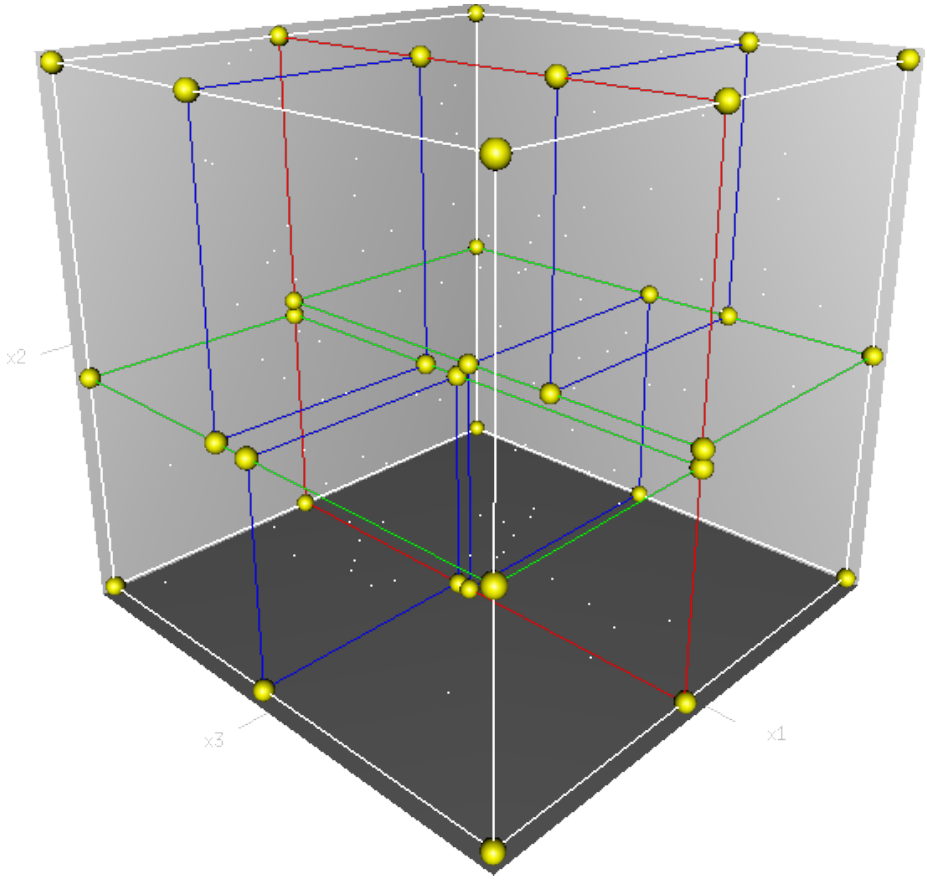

Figura 1: Representación del KD Tree en 3D

1.1. Usos del KD Tree

- Búsqueda ortogonal en un árbol kd: Usar un árbol kd para encontrar todos los puntos que se encuentran en un rectángulo determinado (o análogo de más dimensiones). Esta operación también se denomina rango de búsqueda ortogonal.
- Determinar dónde evaluar una superficie: En las regresiones locales es común evaluar la superficie contenida directamente sólo por los vértices del árbol kd e interpolar en algún punto. Este uso, reflejado en la imagen de arriba, busca asegurar que sólo se realizarán las evaluaciones directas necesarias. Como los árboles kd se .ªdaptan.ªl espacio, este método puede suministrar una excelente aproximación a las verdaderas superficies de regresión local.

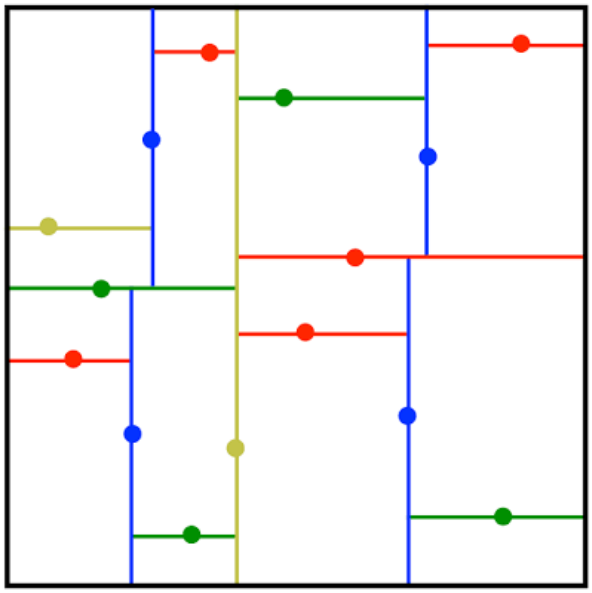

Figura 2: Representación del KD Tree en 2D

2. KD Tree 2D en Python

Usando el lenguaje de programación Python,la librería Pygame para la representación en 2D,la librería Networkx para dibujar nuestro árbol y la librería Matplotlib para poder guardar la imagen de nuestro árbol hemos conseguido representar el KD Tree, aquí las líneas de código para poder crear nuestra ventana gráfica y ver las variables que se usarán para nuestro código :

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3  import time
4  import pygame,sys
5  from pygame.locals import *
6  import random
7
8  #Creamos esta funcion para no confundirnos
9  #ya que pygame tiene las coordenadas del eje Y invertidas
10 def camb(y):
11   return 800-y
12
13 def camb2(x):
14   return x+20
15
16 #Creamos nuestra ventana
17 pygame.init()
18 ventana = pygame.display.set_mode((1000,900))
19 Color2=pygame.Color(255,255,255)
20 pygame.draw.line(ventana,Color2,(0+20,camb(0)),(0+20,camb(800)),10)
21 pygame.draw.line(ventana,Color2,(0+20,camb(0)),(900+20,camb(0)),10)
22 pygame.display.set_caption("Kd-Tree")
23
24 #funte para escribir la coordenada de los puntos
25 fuente = pygame.font.Font(None, 30)
26
27 #En este array guardaremos el punto inicial y final
28 #de donde deberia ir la recta que generan los puntos
29 #el orden para las 3 listas es preorden
30 v=[]
31 #Aqui guardamos los colores que tocan a cada recta
32 co=[]
33 #Aqui guardamos las coordenas de los puntos para
34 #dibujar las coordenadas de los puntos
35 pp=[]
36 #Creaci n de nuestro grafo dirigido
37 G = nx.DiGraph()
38
39 k=2
```

Listing 1: Creacion de ventana y declaración de variables

2.1. Clase Nodo

En esta parte definimos nuetras clase nodo y sus funiones principales hasta build_kdtree.

```
#Nuestra clase nodo
class Node():
  """docstring for Node"""
  def __init__(self, point, axis):
    self.point = point
    self.axis = axis
    self.left = None
    self.right = None
    #Limite mayor en el eje x
    self.Mx = None
    #Limite menor en el eje x
    self.mx = None
    #Limite mayor en el eje y
    self.My = None
    #Limite menor en el eje y
    self.my = None
#Ordenar por y
def orde(tupla):
  return (tupla[1], -tupla[0])

#Ordenar por x
def ord2(tupla):
  return (tupla[0], -tupla[1])

#Aqui construimos el kd tree
def build_kdtree(points, depth=0):
  if not points:
    return None
  #Para saber si dividir por el eje x o y
  axis = depth % k
  if(axis==0):
    points=sorted(points, key=ord2)
  else:
    points=sorted(points, key=orde)

  median = len(points)//2

  node = Node(points[median], axis)

  node.left=build_kdtree(points[:median],depth+1)
  node.right=build_kdtree(points[median+1:],depth+1)

  return node
```

Listing 2: Clase Nodo

2.2. Funciones límite para gráfico

Aquí definimos nuestras funcione límite para saber hasta que parte de la pantalla se han de dibujar nuestras líneas.

```
#Aqui recorremos el arbol en preorden para definir los limites de
    cada punto
def limi(root):
  if(root==None):
    return

  if(root.axis==0):
    if(root.left!=None):
      root.left.mx = root.mx
      root.left.Mx = root.point[0]
      root.left.my = root.my
      root.left.My = root.My
    if(root.right!=None):
      root.right.mx = root.point[0]
      root.right.Mx = root.Mx
      root.right.my = root.my
      root.right.My = root.My
  else:
    if(root.left!=None):
      root.left.mx = root.mx
      root.left.Mx = root.Mx
      root.left.my = root.my
      root.left.My = root.point[1]
    if(root.right!=None):
      root.right.mx = root.mx
      root.right.Mx = root.Mx
      root.right.my = root.point[1]
      root.right.My = root.My

  limi(root.left)
  limi(root.right)

#Aqui tambien recorremos el arbol para sacar los limites y saber
#de donde a donde dibujar la linea divisora
def agreg(root):
  if(root==None):
    return
  if(root.axis==0):
    if(root.left!=None):
      x=root.point[0]
      y=root.left.point[1]
      pi=[[x,y],[root.mx,y]]
      v.append(pi)
      co.append(1)
      aa=root.left.point
      pp.append(aa)
    if(root.right!=None):
      x=root.point[0]
      y=root.right.point[1]
      pi=[[x,y],[root.Mx,y]]
      v.append(pi)
      co.append(1)
      aa=root.right.point
```

```
53        pp.append(aa)
54    else:
55      if(root.left!=None):
56        y=root.point[1]
57        x=root.left.point[0]
58        pi=[[x,y],[x,root.my]]
59        v.append(pi)
60        co.append(0)
61        aa=root.left.point
62        pp.append(aa)
63      if(root.right!=None):
64        y=root.point[1]
65        x=root.right.point[0]
66        pi=[[x,y],[x,root.My]]
67        v.append(pi)
68        co.append(0)
69        aa=root.right.point
70        pp.append(aa)
71
72    agreg(root.left)
73    agreg(root.right)
```

Listing 3: Funciones límite

2.3. Función para generar nuestro árbol

Implementamos esta funcion para poder crear los nodos y las aristas en nuestro propio árbol.

```
1  def grafi(node):
2      if(node==None):
3          return
4      if(node.left!=None):
5          G.add_node(node.point) #creamos un nodo y lo agregamos a
               nuesro grafo
6          G.add_node(node.left.point) #creamos el nodo con el hijo
               izquierdo
7          G.add_edge(node.point,node.left.point) # creamos la arista
               entre ambos
8
9      if(node.right!=None):
10         G.add_node(node.point) #creamos un nodo y lo agregamos a
               nuesro grafo
11         G.add_node(node.right.point) #creamos el nodo con el hijo
               derecho
12         G.add_edge(node.point,node.right.point)# creamos la arista
               entre ambos
13
14   #llamamos recursivamente a sus hijos izquierdo y derecho para
         realizar la misma accion
15     grafi(node.left)
16     grafi(node.right)
```

Listing 4: Función para crear el árbol

2.4. Función Principal

En esta función llamamos a todo lo anterior para poder generar nuesro gráfico:

```
def main():
  #Cremos una lista con valores aleatorios para hacer el kdtree
  points=[]
  for i in range (10):
    x=random.randint(0, 18)
    y=random.randint(0, 16)
    points.append((x,y))
  #points=[(7,2), (5,4), (9,6), (4,7), (8,1), (2,3)]
  #Construimosl arbol
  root=build_kdtree(points)
  #seteamos los limites iniciales despues de conocer la raiz
  root.mx=0
  root.Mx=18
  root.my=0
  root.My=16
  #sacamos los limites
  limi(root)
  #Le mandamos a v los valores para construir la primera linea
  pi=[[root.point[0],0],[root.point[0],16]]
  v.append(pi)
  #Le asignamos a esta el color azul
  co.append(0)
  aa=root.point
  pp.append(aa)
  agreg(root)
  #i sera nuestro iterador de los puntos
  i=0
  grafi(root)
  pos = hierarchy_pos(G,root.point)
  nx.draw(G, pos=pos, node_size=2000,with_labels=True)
  plt.savefig('arbol.png') #guardamos la imagen del arbol
  while True:
    #Hacemos un loop para que la ventana funcione
    for event in pygame.event.get():
      if event.type==QUIT:
        pygame.quit()
        sys.exit()
      #Colocamos este evento para hacer que avance con la tecla
          Left
      if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_LEFT:
          #Cuando ya se hayan recorrido todos los puntos,
              reiniciamos la pantalla
          #Para volverla a mostrar
          if(i==len(v)):
            ventana.fill((0,0,0))
            pygame.draw.line(ventana,Color2,(20,camb(0)),(20,camb
                (800)),10)
            pygame.draw.line(ventana,Color2,(20,camb(0)),(900+20,
                camb(0)),10)
            i=0
          else:
            #Le damos el color que corresponde a nuestra variable
                Color
```

```
50              if(co[i]==0):
51                Color=pygame.Color(0,0,255)
52              else:
53                Color=pygame.Color(255,0,0)
54              #vamos imprimiendo los puntos en consola
55              print(pp[i])
56              #Transformamos el punto en string para mandarlo a
                    pantalla
57              text=" ".join(str(x) for x in pp[i])
58              mensaje=fuente.render(text,1,(0,255,0))
59              #Aqui dibujamos nuestra linea, lo multiplicamos por 50
                    para hacer una escala
60              #Ya que nuestra pantalla es 900*800
61              pygame.draw.line(ventana,Color,(v[i][0][0]*50+20,camb(v
                    [i][0][1]*50)),(v[i][1][0]*50+20,camb(v[i
                    ][1][1]*50)),10)
62              ventana.blit(mensaje,(pp[i][0]*50-15+20,camb(pp[i
                    ][1]*50)))
63              #Aumentamos en 1 nuestor iterador
64              i=i+1
65      pygame.display.update()
```

Listing 5: Función dividir

3. Imagen Ejemplo en 2D

Aqui mostramos una imagen del resultado que obtuvimos al crear KD Tree con 10 puntos aleatorios del 0 al 18 para el eje x y del 0 al 16 para el eje y.

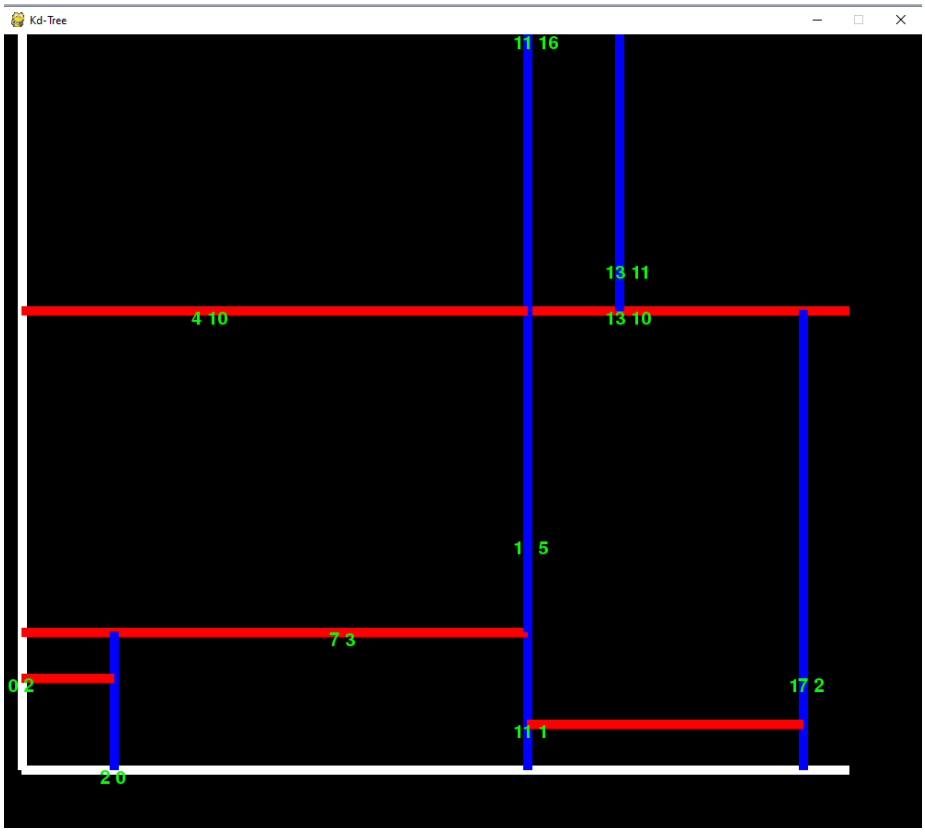

Figura 3: KD Tree con 10 puntos aleatorios

4. Árbol generado para el ejemplo anterior

Mostramos el árbol que nos genera nuestro KD Tree para 10 puntos aleatorios.

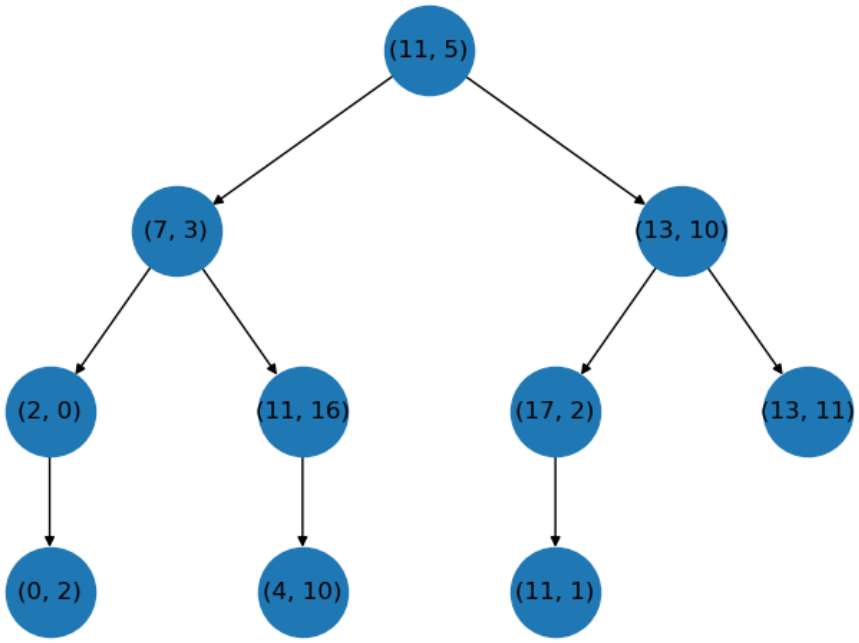

Figura 4: Árbol generado con 10 puntos aleatorios

5. KD Tree 3D en Python

Usando el lenguaje de programación Python,la librería VTK para la representación en 3D,la librería Networkx para dibujar nuestro árbol y la librería Matplotlib para poder guardar la imagen de nuestro árbol hemos conseguido representar el KD Tree, aquí las líneas de código para poder crear nuestra ventana gráfica y ver las variables que se usarán para nuestro código :

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3  import time
4  from vtk import *
5  import random
6
7  #En este array guardaremos el punto inicial y final
8  #de donde deberia ir la recta que generan los puntos
9  #el orden para las 3 listas es preorden
10 v=[]
11 #Aqui guardamos los colores que tocan a cada recta
12 co=[]
13 #Aqui guardamos las coordenas de los puntos para
14 #dibujar las coordenadas de los puntos
15 pp=[]
16
17 G = nx.DiGraph()
18
19 k=3
```

Listing 6: Creacion de ventana y declaración de variables

5.1. Clase Nodo

En esta parte definimos nuetras clase nodo y sus funiones principales hasta build_kdtree.

```
#Nuestra clase nodo
class Node():
  """docstring for Node"""
  def __init__(self, point, axis):
    self.point = point
    self.axis = axis
    self.left = None
    self.right = None
    #Limite mayor en el eje x
    self.Mx = None
    #Limite menor en el eje x
    self.mx = None
    #Limite mayor en el eje y
    self.My = None
    #Limite menor en el eje y
    self.my = None
    #Limite mayor en el eje z
    self.Mz = None
    #Limite menor en el eje z
    self.mz = None
```

Listing 7: Clase Nodo

5.1.1. Funciones de la clase nodo

```
#Ordenar por y
def orde(tupla):
  return (tupla[1])

#Ordenar por x
def ord2(tupla):
  return (tupla[0])

#Ordenar por z
def ord3(tupla):
  return (tupla[2])


#Aqui construimos el kd tree
def build_kdtree(points, depth=0):
  if not points:
    return None
  #Para saber si dividir por el eje x, y o z
  axis = depth % k
  if(axis==0):
    points=sorted(points, key=ord2)
  elif(axis==1):
    points=sorted(points, key=orde)
  else:
    points=sorted(points, key=ord3)

  median = len(points)//2

  node = Node(points[median], axis)

  node.left=build_kdtree(points[:median],depth+1)
  node.right=build_kdtree(points[median+1:],depth+1)

  return node
```

Listing 8: Funciones de la clase nodo

5.2. Funciones límite para gráfico

Aquí definimos nuestras funcione límite para saber hasta que parte de la pantalla se han de dibujar nuestras líneas.

```
#Aqui recorremos el arbol en preorden para definir los limites de
    cada punto
def limi(root):
  if(root==None):
    return

  if(root.axis==0):
    if(root.left!=None):
      root.left.mx = root.mx
      root.left.Mx = root.point[0]
      root.left.my = root.my
      root.left.My = root.My
      root.left.mz = root.mz
      root.left.Mz = root.Mz
    if(root.right!=None):
      root.right.mx = root.point[0]
      root.right.Mx = root.Mx
      root.right.my = root.my
      root.right.My = root.My
      root.right.mz = root.mz
      root.right.Mz = root.Mz
  elif(root.axis==1):
    if(root.left!=None):
      root.left.mx = root.mx
      root.left.Mx = root.Mx
      root.left.my = root.my
      root.left.My = root.point[1]
      root.left.mz = root.mz
      root.left.Mz = root.Mz
    if(root.right!=None):
      root.right.mx = root.mx
      root.right.Mx = root.Mx
      root.right.my = root.point[1]
      root.right.My = root.My
      root.right.mz = root.mz
      root.right.Mz = root.Mz
  else:
    if(root.left!=None):
      root.left.mx = root.mx
      root.left.Mx = root.Mx
      root.left.my = root.my
      root.left.My = root.My
      root.left.mz = root.mz
      root.left.Mz = root.point[2]
    if(root.right!=None):
      root.right.mx = root.mx
      root.right.Mx = root.Mx
      root.right.my = root.my
      root.right.My = root.My
      root.right.mz = root.point[2]
      root.right.Mz = root.Mz

  limi(root.left)
```

```
53    limi(root.right)
54
55  #Aqui tambien recorremos el arbol para sacar los limites y saber
56  #de donde a donde dibujar la linea divisora
57  def agreg(root):
58    if(root==None):
59      return
60    if(root.left!=None):
61      pi = plane(root.left)
62      v.append(pi)
63      aa=root.left.point
64      ap=Punto(aa,root.axis)
65      pp.append(ap)
66    if(root.right!=None):
67      pi = plane(root.right)
68      v.append(pi)
69      aa=root.right.point
70      ap=Punto(aa,root.axis)
71      pp.append(ap)
72    agreg(root.left)
73    agreg(root.right)
```

Listing 9: Funciones límite

5.3. Función para generar nuestro árbol

Implementamos esta funcion para poder crear los nodos y las aristas en nuestro propio árbol.

```
1   def grafi(node):
2       if(node==None):
3           return
4       if(node.left!=None):
5           text=" ".join(str(x) for x in node.left.point)
6           text2=" ".join(str(x) for x in node.point)
7           G.add_node(text)
8           G.add_node(text2)
9           G.add_edge(text2,text)
10      if(node.right!=None):
11          text=" ".join(str(x) for x in node.right.point)
12          text2=" ".join(str(x) for x in node.point)
13          G.add_node(text)
14          G.add_node(text2)
15          G.add_edge(text2,text)
16      grafi(node.left)
17      grafi(node.right)
```

Listing 10: Función para crear el árbol

5.4. Clase Punto

En esta función creamos la estructura gráfica punto, que será usada para insertar y visualizar los puntos en nuestra ventana.

```
class Punto():
  """docstring for Punto"""
  def __init__(self, l, axis):
    self.l=l
    if(axis==0):
      self.color=[0,0,255]
    elif(axis==1):
      self.color=[255,0,0]
    else:
      self.color=[0,255,0]

    self.punt = vtkSphereSource()
    self.punt.SetRadius(1)
    self.punt.SetCenter(l[0],l[1],l[2])

    self.puntMapper = vtkPolyDataMapper()
    self.puntMapper.SetInputConnection(self.punt.GetOutputPort())

    self.puntActor = vtkActor()
    #self.puntActor.GetProperty().SetColor(self.color)
    self.puntActor.SetMapper(self.puntMapper)
```

Listing 11: Clase Punto

5.5. Clase Plano

En esta función creamos la estructura gráfica plano, que será usada para insertar y visualizar los planos en nuestra ventana.

```
class plane():
  def __init__(self, root):
    self.cube = vtkCubeSource()
    if(root.axis==1):
      self.color=[255,0,0]
      self.cube.SetYLength(0.5)
      self.cube.SetXLength(root.Mx-root.mx)
      self.cube.SetZLength(root.Mz-root.mz)
      self.cube.SetCenter(float((root.Mx+root.mx)/2),root.point[1],
          float((root.Mz+root.mz)/2))
    elif(root.axis==2):
      self.color=[0,255,0]
      self.cube.SetYLength(root.My-root.my)
      self.cube.SetXLength(root.Mx-root.mx)
      self.cube.SetZLength(0.5)
      self.cube.SetCenter(float((root.Mx+root.mx)/2),float((root.My
          +root.my)/2),root.point[2])
    else:
      self.color=[0,0,255]
      self.cube.SetYLength(root.My-root.my)
      self.cube.SetXLength(0.5)
      self.cube.SetZLength(root.Mz-root.mz)
```

```
21          self.cube.SetCenter(root.point[0],float((root.My+root.my)/2),
                float((root.Mz+root.mz)/2))
22      self.cubeMapper = vtkPolyDataMapper()
23      self.cubeMapper.SetInputConnection(self.cube.GetOutputPort())
24      self.cubeMapper.SetResolveCoincidentTopologyToShiftZBuffer()
25
26      self.cubeActor = vtkActor()
27      self.cubeActor.SetMapper(self.cubeMapper)
28      self.cubeActor.GetProperty().SetColor(self.color)
29      self.cubeActor.GetProperty().SetOpacity(0.5)
```

Listing 12: Clase Plano

5.6. Función principal

En esta función creamos nuestro ejes x,y,z con planos de referencia y construimos nuestro KD Tree con 10 puntos aleatorios respceto a cada eje. Podemos darnos cuenta que en la línea 103 definimos nuesra funcion KeyPress puesto que esta hace que al momento de presionar la tecla left pueda dibujar los planos en nuestra ventana y avance consecutivamente el gráfico hasta que dibuje todos los planos respectivos.

```
1
2   #Cremos una lista con valores aleatorios para hacer el kdtree
3   points=[]
4   for i in range (10):
5     x=random.randint(0, 100)
6     y=random.randint(0, 100)
7     z=random.randint(0, 100)
8     points.append([x,y,z])
9   #points=[(7,2,1), (5,4,2), (9,6,3), (4,7,4), (8,1,5), (2,3,6)]
10  #Construimosl arbol
11  root=build_kdtree(points)
12
13  grafi(root)
14  text=" ".join(str(x) for x in root.point)
15  pos = hierarchy_pos(G,text)
16  nx.draw(G, pos=pos, node_size=2000,with_labels=True)
17  plt.savefig('arbol.png')
18
19  #seteamos los limites iniciales despues de conocer la raiz
20  root.mx=0
21  root.Mx=100
22  root.my=0
23  root.My=100
24  root.mz=0
25  root.Mz=100
26  #sacamos los limites
27  limi(root)
28  #Le mandamos a v los valores para construir la primera linea
29  pi=plane(root)
30  v.append(pi)
31  #Le asignamos a esta el color azul
32  aa=root.point
33  ap=Punto(aa,0)
```

```
pp.append(ap)
agreg(root)
print(len(v))
#i sera nuestro iterador de los puntos
i=0

ren = vtkRenderer()
ren.SetBackground(0, 0, 0)

color=[255,255,255]
cube = vtkCubeSource()
cube.SetYLength(0.5)
cube.SetXLength(100)
cube.SetZLength(100)
cube.SetCenter(50,0,50)

cubeMapper = vtkPolyDataMapper()
cubeMapper.SetInputConnection(cube.GetOutputPort())
cubeMapper.SetResolveCoincidentTopologyToShiftZBuffer()
cubeActor = vtkActor()
cubeActor.SetMapper(cubeMapper)
cubeActor.GetProperty().SetColor(color)
cubeActor.GetProperty().SetOpacity(0.5)

cube2 = vtkCubeSource()
cube2.SetYLength(100)
cube2.SetXLength(0.5)
cube2.SetZLength(100)
cube2.SetCenter(0,50,50)

cubeMapper2 = vtkPolyDataMapper()
cubeMapper2.SetInputConnection(cube2.GetOutputPort())
cubeMapper2.SetResolveCoincidentTopologyToShiftZBuffer()
cubeActor2 = vtkActor()
cubeActor2.SetMapper(cubeMapper2)
cubeActor2.GetProperty().SetColor(color)
cubeActor2.GetProperty().SetOpacity(0.5)

cube3 = vtkCubeSource()
cube3.SetYLength(100)
cube3.SetXLength(100)
cube3.SetZLength(0.5)
cube3.SetCenter(50,50,0)

cubeMapper3 = vtkPolyDataMapper()
cubeMapper3.SetInputConnection(cube3.GetOutputPort())
cubeMapper3.SetResolveCoincidentTopologyToShiftZBuffer()
cubeActor3 = vtkActor()
cubeActor3.SetMapper(cubeMapper3)
cubeActor3.GetProperty().SetColor(color)
cubeActor3.GetProperty().SetOpacity(0.5)

ren.AddActor(cubeActor)
ren.AddActor(cubeActor2)
ren.AddActor(cubeActor3)

for punti in pp:
```

```
  ren.AddActor(punti.puntActor)



renWin = vtkRenderWindow()
renWin.AddRenderer(ren)
renWin.SetWindowName("Kd-Tree")
renWin.SetSize(500,500)

iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

def KeyPress(obj,event):
  global v
  global i
  global pp
  key = obj.GetKeySym()
  if(key=="Left"):
    if(i < len(v)):
      ren.AddActor(v[i].cubeActor)
      pp[i].puntActor.GetProperty().SetColor(v[i].color)
      pp[i].punt.SetRadius(1.5)
      print(pp[i].l)
      i=i+1
      iren.Render()
    else:
      for act in v:
        ren.RemoveActor(act.cubeActor)
      for act in pp:
        act.puntActor.GetProperty().SetColor([255,255,255])
        act.punt.SetRadius(1)
      i=0
      print("-------------------------------------------------")
      iren.Render()


iren.SetInteractorStyle(vtk.vtkInteractorStyleTrackballCamera())
iren.AddObserver("KeyPressEvent", KeyPress)

iren.Initialize()
iren.Start()
```

Listing 13: Función principal

6. Imagen Ejemplo en 3D

Aqui mostramos una imagen del resultado que obtuvimos al crear KD Tree con 20 puntos aleatorios del 0 al 100 para los 3 ejes .

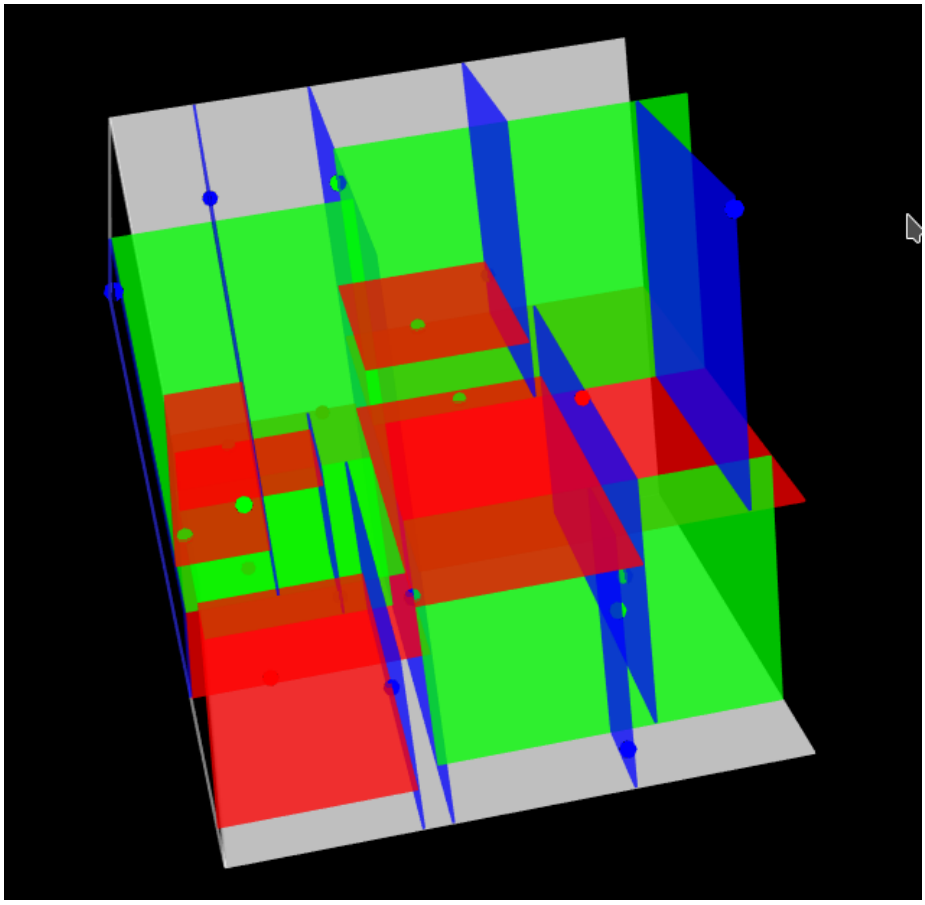

Figura 5: KD Tree con 20 puntos aleatorios

7. Árbol generado para el ejemplo anterior

Aqui mostramos el árbol generado por nuestro KDTree en el caso probado anteriormente.

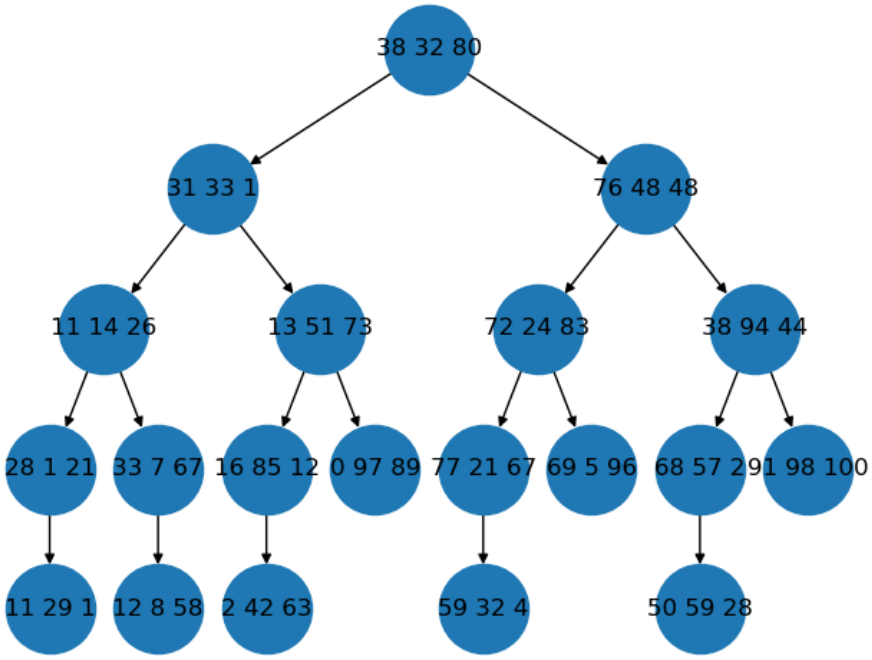

Figura 6: Árbol generado con 20 puntos aleatorios

8. Conclusiones

- Nos dimos cuenta que al igual que en el Quadtree y Octree esta estructura nos puede ayudar mucho al momento de querer distribuir datos de una manera no uniforme, ya que pueden estar dispersos en cualquier espacio donde nosotros lo definamos y querramos que esten.
- Vimos que hay una similitud con respecto al Quadtree y Octree porque en ambos casos uno puede ser en 2D y en 3D respectivamente pero en este caso puede adaptarse para esas dimensiones y para más si en caso hubiera.
- Nos dimos cuenta que en el KDTree es mucho más fácil el almacenamiento de los datos respecto a las anteriores estructuras trabajadas.

9. Referencias

- Árbol kd. (2019). Retrieved 13 November 2019, from https://es.wikipedia.org/wiki/
- K Dimensional Tree — Set 1 (Search and Insert) - GeeksforGeeks. (2019). Retrieved 13 November 2019, from https://www.geeksforgeeks.org/k-dimensional-tree/
- VTK/Examples/Python/GeometricObjects/Display/Cube - KitwarePublic. (2019). Retrieved 13 November 2019, from https://vtk.org/Wiki/VTK/Examples/Python/GeometricObjects/Display/Cube
- Pygame. (2019). Retrieved 13 November 2019, from https://www.pygame.org/news